PYTHON

CONTENT

1. JUPYTER & CONDA
2. FIRST DATASET EXPLORATIO
3. VARIABLES AND DATA TYPES
4. CONTROL FLOW AND ITERATION (for, if, elfe, while)
5. LIST COMPREHENSION
6. FUNCTIONS
7. CLASSES AND OBJECTS
8. PARAMETER PASSING
9. HELP IN PYTHON
10. Visual Studio Code (IDE)
11. DIRECTORIES AND FILES
12. PRINT
13. PYTHONIC CODE

1. JUPYTER & CONDA

VIRTUAL ENVIRONMENTS

*** ipython:** most popular version of a python has magic commands to operate with the system starting witH % and %%

*** Virtual Env:** an isolated copy of Python that maintains its own files and directories. Used to ensure that each project is cleanly separated and there are no problems with dependencies between them.

*** Conda:** a package and environment manager that we can use to create environments.

*** .yml environment file** list of packages with their version number. Required to create conda env. Eg:

```
name: conda_env1
channels:
  - anaconda
  - conda-forge
dependencies:
  - python=3.6
  - numpy=1.15
```

*** Conda vs Docker;** Docker is ranked 7th while Conda is ranked 15th. The most important reason people chose Docker is that the Docker creates a single object, containing an app. with all dependencies, that can be moved between any docker-enabled machines**. FOR MORE:**

https://www.slant.co/versus/1592/5880/~conda_vs_docker

CREATE THE ENVIRONMENT

*** [1]. List existing conda environments**

OS, Linux: open a Terminal.

Win: open Anaconda command prompt from the Start menu.

```
> conda env list
# conda environments:
# base                  *  C:\Users\user\anaconda3
```

Note: If you didn't previously install any Conda env, it should only display the default one called base, or sometimes root, along with its installation path which may vary depending on your machine and whether you installed Anaconda or Miniconda.

Troubleshooting: If this command doesn't work f or you, then it's likely that your terminal doesn't find conda. In that case, verify that you have Anaconda or Miniconda installed

*** [2]. Create the env, with an existing env file**

```
> conda env create -f conda_env1
# you need to be in the same folder as the exts-ml.yml
> conda env create -f C:\Users\your_username\Downloads\exts-ml.yml  #
Windows
> conda env create -f /Users/your_username/Downloads/exts-ml.yml
# OS
```

Note: It may take a while - all packages are being downloaded form internet!, sometimes several GB

Troubleshooting: If you cannot locate your .yml file from the terminal or the Anaconda prompt, use the File Explorer to find its absolute path. Open the explorer and go to the file (ex. in your Downloads folder), right click on it and open the details window. You should be able to copy/paste the absolute path from there and use it in the command:

[3]. Test the env.

Repeat [1] and see whether you have new env

You may run it for a test: conda activate exts-ml

ACTIVATE CONDA ENVIRONMENT

*** [1]. Activate / Deactivate**

```
conda activate exts-ml
conda deactivate
# new line should start with:
(exts-ml) pawels-MacBook-Pro-2:Resourses pawel$
```

*** [2]. Older version**

```
source activate exts-ml
jupyter lab
# jupyter is my envireoment, must be downloaded as package for
. that envireoment
```

*** [3]. Test using iphython in that env.**

```
> ipython. # ipython version
> import seaborn as sns
> data = sns.load_dataset('titanic') # load titanic data from gitgub
> data.head(5)
> exit()
```

2. HELP IN PYTHON

ONLINE

- **Python documentation:**

https://docs.python.org/3/

- **Tutor mailing list**

https://mail.python.org/mailman/listinfo/tutor $

- **StackOverflow**

https://stackoverflow.com/questions/tagged/python

- **Quora**

https://www.quora.com/topic/Python-programming-language-1?merged_tid=13292

BUILD-IN

*** help()**

Python has a built-in function called help() that can return documentation on any object, method or attribute.

For example typing in a Jupyter notebook help(float)returns an entire documentation for the object type float, and the information is displayed directly in the notebook. We can similarly inquire about any function. For example, we can inquire about the abs() function as follows

*** function_name?**

We can also use the question mark symbol to access the source code of a function or feature. This can help give you some quick insight into how the function is actually implemented. For example, we can inquire about the function len() as follows.

eg: len?

*** SHIFT + TAB**

Another way to obtain help is using the Jupyter notebook's built-in **shortcut shift + tab**. If we simultaneously press the shift and tab keys, we obtain information about the object that we just typed inside the cell. The advantage

3. Visual Studio Code (IDE)

*** Get Visual studio code**

Abbrev: VScode

Go to: www.code.visualstudio.com

Why to use: free, run on IOS, Windows and Linux

*** Instal**

1. Click on extension. Icon
   and search for python in search box
2. Instal (if not already in a package)
   Linting, Debugging (multi threaded remote… Microsoft plug in)
   it adds support for debugging
   it is not essential
   click on it to see rating and functions

*** Set up**

1. Open the folder with some .py files
   top left icon
   open any .py file
2. load, and oped debugger (left site button with a bug)
3. Go to: DEBUG (top.left, there is no configurations)
   here you must go to folders and create
   (open) new folder
   to make any changes in configs
   choose: python environment
   this brings laugh Jason file
   modify Jason file
   go to: "cwd"
   remove: "{bla-bla something}"
   keep "" empty string
   to ensure that debugger has the same directory as our working directory with the file being debugged. I don't have it so keep it in mind that it may save stuff somewhere !!!

4. **PROBLEM:** Having more than one version of Python
   go to: code menu; preferences; settings
   search "python" "path"; in my case it is:
   "python.pythonPath":
   this is the command that is executed by VScode
   when using python (on the right site, of settings in brackets)
   it can be: "python.pythonPath": "python3"

*** VCstudio Useful Commands**

*** STOP:** control + C or command + C I don't remember

*** FONT +** command + (Command + Shift + 1/1)

*** FONT -** command -

 # many Lines Command + Shift + 7/"/"

click: gear icon

3. VARIABLES AND DATA TYPES

VARIABLES AND DATA TYPES

In Python, a variable is a way of referring to a memory location.

VARIABLES DO NOT HAVE ASSOCIATED TYPES: a major difference from Java and C/ C++.

DATA TYPES

1. **number,** with four different subtypes:
   * **int:** for storing integers
   * **float:** for storing decimal numbers
   *__complex:__ complex numbers, real + imaginary part, **5+2j**
2. **string:** for storing sequences of Unicode characters
3. **List:** ordered list of values
4. **tuple:** same as lists, but immutable
5. **dictionary:** for storing a list of values that have an index

DATA STRUCTURES

1. **Lists** # [ ]; mutable; List = [1, 2, 3, 4, 5, 6]
2. **Tupples** # ( ); immutable; Tupple = (1, 2, 3, 4, 5, 6)
3. **Dictionaries** # { } ; key: value pairs;
   Dict = { "one": 1, "two": 2, "three": 3 }
4. **Set** # { }; unordered list of unique values;
   Set = {1, 2, 3, 4, 5, 6}
5. **List comprehension** # [ ],{ }; like lambda but easier to use

STRING

https://docs.python.org/3/library/string.html#string.ascii_lowercase

S.FEATURES

+ **immutable**; new object with new id(), (like in Java and C#)
+ **slicing:** substrings can be created with the slicing notation
+ **Quotas** ‘ ’;“ ”;“” “”; “”” “””; **almost no differences:**
  > **single quotes** are used by most python users !!!
  > **double quotes inside single quotes** & opposite
  without escaping them – ie these will be printed
  > str with **triple quotes** can span over many lines

S.INSPECT

* **len()**
* **in;** s = "aaaYaaa"; "Y" in s # True
* **s.startwith( );** True if the string starts with PATTERN
* **s.endwith( );** True if the string starts with “example”

S.PRINT

- **.format()**
  ”{} plus {} equals to **{}".format(1, 2, 3);**
  # prints: 1 plus 2 equals to 3
- **print( )**
  print('string is {}'.format(str)) or print(str)

S.MODIFY

* **(+):** concatenation; strings can be glued together: **“a” + “b”** ==“ab”
* **(*)** repetition; repeatedly concatenated **“a”*40** ‘aaaaaa….aaaaaa’
* **replace( );** s.replace(“e”, “wow”); replace “e” with “wow” in string s.
* **split( );** s.split(“tt”), splits string s into a list of strings, on “tt“.
* **capitalize( );** x = "seven".capitalize(); print("x is {}".format(x));
* **join( );**

```
x = [1, 2, 3, 4, 5, 6, 7 ];
','.join(str(x) for i in x)
#       '1,2,3,4,5,6,7'
#       use list comprehencion to transform all items in list     #
```

BOOL

* **True:** -> **True**
  -> **1**
  -> **or any number !=0** e.g. -4000 and 3.4565

* **False:** -> **False**
  -> **0**
  -> **None**
  -> **string with anything or empty sting**

TUPLE

+ **TUPLES:** like lists except that they cannot be changed. To distinguish them from lists, tuples are defined using parentheses as follows

tupple_example = (1, 2, 3)

THE REST IS LIKE IN LIST

LIST

L.FEATURES

List: the most versatile data type, a collection of items separated by commas and enclosed in square brackets.

1. ordered
2. indexing starts at 0
3. they can contain objects of any data type
4. can have any size or empty
5. the elements of a list can be changed, unlike in tupple
6. lists can contain sub lists and they can be arbitrarily nested

L.EXAMPLES

- **list = [2.34, 10, 'John', 'car', 9];** different dtypes
- **list = [ ];** empty list
- **list = [['dog' , 23] , 10];** list in a list

MAKE NEW LIST

- **[1, 2, "string"]**
- **list( )**
- **list(range( ))**
- **new_list = old_list.copy( );** one level only

L.POPULAR FUNCTIONS

* **List.append("new_item")**
* **" ".join([List])**

L.ACCESS ITEMS

* **L[beginning: end: step],** **INDEXING -** end is not included
* **L[::-1]** # reverse complement
* **L.reverse();** Reverse the elements of the list, in place.
  Like reverse complement

L.INSPECT

* **isinstance( )** isinstance(L, list) # True/False
* **len( )** ; list lenght
* **L.index(x)** # to find index of an item in list;
  ["a", "b", "c"].index("a"); # 0
* **L.count( )** # number of times string appears in the list
  ["a", "b", "c"].count("a"); # 1
* **enumerate(L)** look what is inside the list
  days = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]
  for i, d in enumerate(days): print(f"day {I} is {d}")

L.ADD OR MODIFY ITEM

* **L[i] = "newItem"** # replace an item with index i
* **L .append( )** # add new item to list at its end.
  Caution may create list inside list
  L1 = ["a"] ; L2 = ["b", "c"]
  L1.append(L2) # L1 == ["a", ["b", "c"]]
* **List.extend( )**
  # same as append but **adds every item separately**,
  L1 = ["a"] ; L2 = ["b", "c"]
  L1.extend(L2) # L1 == ["a", "b", "c"]
* **=+** same as l.extend(), can not be used with pipe "I"
* **+** Add two lists
  L1 = ["a"] ; L2 = ["b", "c"]
  L1 + L2 == ["a", "b", "c"]
* ***** Multiply the list; L1 * 3; ["a", "b", "c", "a", "b", "c", "a", "b", "c"]
* **L.insert( )**; insert an item x at a given position i
  L.insert(i, x); Insert an item x at a given position i
  i indicate index of the element before which to insert!
  List.insert(0, x) # inserts at the front of the list
  List.insert(len(l), x) # like: a.append(x), add at the end!

L.REMOVE ITEM

* **L.clear( )** # clears an entire list
* **L.pop([i])** # remove last el, or indexed el, returns it.
* **L.remove(x)** # x is an item selected,
  . an exact name of the item must be given!
* **del** # del items in list using its index, NO Brackets!
  del listExample[0:3]

SORT ITERABLE

* **L.sort( )** # as in sorted()
* **sorted( )** # sorted(iterable[, cmp[, key[, reverse]]]
+ **LIST** /1/. sorted([5, 2, 3, 1, 4]) # [1, 2, 3, 4, 5]
  /2/. a = [5, 2, 3, 1, 4]; a.sort(); a; [1, 2, 3, 4, 5]
+ **DICT** sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'});
  # [1, 2, 3, 4, 5]
+ **STR** sorted("This is a test string".split(), key=str.lower)
  # ['a', 'is', 'string', 'test', 'This']

SORT COMPLEX OBJ

+ **EG:** **use object's indices as keys**

```
student_tuples = [ ('john', ' A', 15), ('jane', 'B', 12), ('dave', 'B', 10) ]
sorted(student_tuples, key=lambda student: student[2])
#  [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]  # sorted by age
```

LIST FOR STACKS

The list methods make it very easy to use a list as a stack, where the last el. added is the 1st el. retrieved ("**last-in, first-out**"). To add an item to the top of the stack, use **l.append().** To retrieve an item from the top of the stack, use **l.pop()** without index – it's super fast. Doing inserts or pops from the beginning of a list is slow (because all of the other el's have to be shifted by one).

DICTIONARY

FEATURES

- A searchable list of **key-value pairs**
- **Every key is unique,** values can be duplicated
- **Hashed array** in other languages
- Index doesn't work - **use Keys** instead
- order is not important – no sorting

Caution: add key:value pairs, if you use preexisting key, its value will be removed, and new added.

NEW DICT

- **dict( )** dct = dict(); Empty dict
- **dict(key = value)** dict(one = 1, two = 2, three = 3)
- **{ key : vale }** { "one": 1, "two": 2, "three": 3 }
- **{**unpacked.dct}** dct3 = { **dct1, **dct2 }

ADD OR MODIFY ITEMS

*** dct[Key] = value** Used to add new key : value pair or to update value with pre-existing key

- dt ["a"] = 11
- key, value = "key1", "value1"
  dt[key] = value

*** dct.update({ Key : Value })** dct.update({"a": 11})

+ one key + multiple values; only the last value will be used

- dct = { "one": 1, "one": 2, "one": 3}; dct
  # {'one': 3}

+ join dictionaries; UNPACK dict before joining

- dct3 = { **dct1, **dct2 }

+ Duplicated keys in joined dct; the value from the last entry of a given key will be used to build a dict!

INSPECT & ACCESS

*** dct["key"]** InSquared Brackets, retrieve a value by its key **x**

*** d.get( );** Used to Avoid KeyError if you use non-exiting key, eg:

```
dct = { "one": 1, "two": 2, "three": 3 }
dct["four"]          #   KeyError: 'four'
dct.get("four")      #   None
```

*** d.keys( );** access the keys only

```
for k in dct.keys(): print( f' { k } ' , end="; )
#   one; two; three;
```

*** d.values( );** access the values only

```
for v in dct.values(): print( f' { v } ' , end="; )
#   1; 2; 3;
```

d.items(); Returns key - value pairs,

```
for k, v in dct.items():
        print(f'key is {k}, value is {v}', end="; "),
```

*** sorted( );** sorts only keys, not values **– not used**

```
sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
#   [1, 2, 3, 4, 5]
```

FIND KEYS

*** in / not in;** **searches KEYs,** not values !!!

```
dct = { "one": 1, "two": 2, "three": 3 }
print("two" in dct ) #    True
```

FOR LOOP WITH DCT

+ Keys only INDEX calls keys, not values !

```
dct = { "one": 1, "two": 2, "three": 3 }
for i in dct: print(i, end=",,,"),
#   one,,, two,, ,three,,,
```

+ Keys & Values; You must call values using each key

```
for i in dct: print( f'{ i } is { dct[ i ] }', end="; ")
#   one is 1; two is 2 ; three is 3
```

SHALLOW COPY vs DEEP COPY

DEF

+ Shallow Copy (Sc) - only one level deep – Sc allows constructing new collection object and then, populating it with **references to child obj's** found in original obj.

+ Deep Copy (Dc); **makes the copying process recursive**. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. Ie. walks the whole object tree to create a **fully independent clone** of the original object and all of its children.

COPY COLLECTIONS

*** =** **factory copy function;** works only for Python built-in mutable collections like **lists, dicts,** and **sets** - doesn't work for custom obj's! eg: new_list = list( ); new_dict = dict( ); new_set = set( )

*** copy.copy(); Creates Sc;** require copy package

```
> import copy;
> List = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
> ShallowCopy_List = copy.copy(List)
> ShallowCopy_List[ 0 ][ 1 ] = 999
        #   changes in original and copied objects
> List  #   [[1, 999, 3], [4, 5, 6], [7, 8, 9]]
```

+ copy.deepcopy() for creating Dc

```
> DeepCopy_List = copy.deepcopy( List )
# done on all levels, the above example would only
affect the copy
```

4. CONTROL FLOW AND ITERATION

IF; ELIF; ELSE

1. the statement must start with if,

2. then it can have only other if or elif.

2. The "else" can be place at the end, but it doesn't have to be

[1]
```
if x < 0: print('x is negative')
elif  x == 0: print('x is zero')
else: print('x is positive')
```

[2]
```
if x < 0: print('x is negative')
if x >= 0: print('x is positive or zero')
```

[3]
```
if x < 0: print('x is negative')
elif  x == 0: print('x is zero')
elif x > 0: print('x is positive')
```

WHILE

*** while;** dangerous functions, if you run it wihtout control, just restart python kernel in jupyter notebook

```
n = 10
while n > 0:
    n = n-1
    print( n, end=": ")
#   9: 8: 7: 6: 5: 4: 3: 2: 1: 0:
```

FOR

+ The iteration in for is always done over the items of a sequence, not like in C ior other languages, where we define step size and a seq,

*** Eg 1.**
```
friends = ['Sophie', 'Ben', 'Lilie']
for friend in friends:
    print(friend, end="; ")
# Sophie; Ben; Lilie; [0, 1, 2] has: {0; 1; 2; }
```

*** Eg 2.**
```
numbers = [[0, 1, 2],[3, 4, 5]]
for x in numbers:
    print(x, "has", end=": {")
    for y in x:
        print(y, end="; ")
    print("} \n")
#  [0, 1, 2] has: {0; 1; 2; }
# [3, 4, 5] has: {3; 4; 5; }
```

BREAK & CONTINUE

*** break; stop the loop** : terminates the loop containing immediately after the body of the loop. If found **inside a nested loop**, then the break terminates the innermost loop only. n = 10

```
while n>0:
    n = n-1
    if n == 5: break
    print(n, end="; ")
#   9; 8; 7; 6;
```

*** continue;** terminates the loop for the current iteration only and returns control of the program to the top of the loop. n = 10

```
while n>0:
    n = n-1
    if n == 5: continue
    print(n, end="; ")
#   9; 8; 7; 6; 4; 3; 2; 1; 0;
```

+ Nested loops with break and continue

[1]
```
for i in range(3):
    print(f'{ i }', end = "=( ")
    for j in range(2):
        print(f'{ j }', end = " / ")
        break
    print(" )",end="; ")
#   0=( 0 /  ); 1=( 0 /  ); 2=( 0 /  );
```

[2]
```
for i in range(3):
    print(f'{ i }', end = "=( ")
    break
    for j in range(2):
        print(f'{ j }', end = " / ")
    print(" )",end="; ")
#   0=(
```

[3]
```
for i in range(3):
    print(f'{ i }', end = "=( ")
    continue
    for j in range(2):
        print(f'{ j }', end = " / ")
    print(" )",end="; ")
#   0=( 1=( 2=(
```

[4] if no breaks or continue was appolied the results would be:
```
#   0=( 0 / 1 /  ); 1=( 0 / 1 /  ); 2=( 0 / 1 /  );
```

ELSE IN FOR LOOP

*** else:** http://book.pythontips.com/en/latest/for_-_else.html

- The ***else clause*** executes after the loop completes normally, ie. the loop did not encounter the break statement.
- ***When it is useful?*** Loops are often used to search for something, and break when it is found. Thus, else in for loop can be used as a flag to notify that all elements were searched, and nothing was found.

```
for i in range(7)
    if i >= 7 : break
    print( i, end="; " )
else:
    print( "there was no more than 7 in the loop")
#   0; 1; 2; 3; 4; 5; 6;
#   there was no more than 7 in the loop
```

IMPORTANT:

else is on the same indentation level as for statement

SMART IF ELSE IN DEF:

```
def test_x( * , x ): # * forces to call par mane in the function
    if (x==0): return f"x is zero look:{x}";
    if (x<0): return f"x is less then zero look:{x}";
    if (x>0): return f"x is more than zero look:{x}";

test_x( x = 200 )
#   'x is more than zero look:200'
```

5. LIST COMPREHENSION (LC)

FEATURES

Used like lambda function to built a temporary function over elements of an iterable. LC uses the for loop to go over elements of an iterable (like a list, array, etc.), giving you access to its elements (x). Additionally, LC are used to make decision, with a conditional operator, eg: if, or to transform values in an iterable

LC

*** [x for x in seq]**

*** { x.key : x.value for x in seq }**

- *** seq** # an itterable with data
- *** x** # operations each data point
- *** for x in** # assignment made by LC
- *** if x == 0** # conditional assignment

LC + IF

*** [x*modification for x in List if condition == True]**

```
old_list =  [0, 5, 10, 15, 20]
new_list = [x*50 for x in old_list if x>5 ]
new_list
#           [500,  750, 1000]
```

LC + IF ELSE

*** [x*modifi1 if condition == True else x*modifi 2 for x in List]**

```
old_list =  [0, 5, 10, 15, 20]
new_list = [x*50 if x>5 else "small_nr" for x in old_list]
new_list
#           ['small_nr', 'small_nr', 500, 750, 1000]
```

RESTULTS IN DIFFERENT OBJECTS

+ LIST **lc = [x for x in seq]**

lc results is a list

+ TUPPLE **lc = [(x, x*2) for x in seq]**

lc results is a list with tupples, build from each x

+ DICT **lc = { x.key : x.value for x in seq }**

lc results is a dict

+ SET **lc = { x.value for x in seq }**

lc is a set!

EXAMPLES

+ RESULTS STORED IN LIST

```
newSeq= [x*2 for x in range(3)]:
newSeq:
#       [0, 2, 4]
```

+ RES. STORED IN TUPLE

if you wish to have the whole object is a tuple, the function **tupple()** must be used, otherwise you get a list with tupples and all new x's in each of them

[1]

```
newSeq= [ (x*16) for x in range( 3 ) ]
newSeq2 = tupple(newSeq)
```

[2]

```
newSeq = [ (x, x*2, x**2) for x in range( 3 ) ]
#       [(0,0,0), (1,2,1), (2,4,4)]
```

+ RES. STOIRED IN DICTIONARY

- lc must be created using { } brackets,
- you need top create key:value pairs for each x in seq

```
{ x:x*20 for x in list(range(3))}
#       {0: 0, 1: 20, 2: 40}
```

+ RESULTS IN SET,

- set is created using { }
- you give only values based on each value in x,

```
{ x*20 for x in list(range(3))}
#       {0, 20, 40}
```

```
#       List with Tupples!
```

newSeq= { x : x*2 for x in range(1, 5)}

6. FUNCTIONS

KEY STATMENTS

*** def:** defines a function. Followed by **function name** and the **function parameters** inside the parentheses, and **a colon**. The statements are **indented**.

*** return** Tells python to exit a function. **A function can have one or more or no return statements.** (i) **return statement without any expression** returns special value **None** (ii) **no return statement**, the function ends at the end of the function body.

*** yield** **Returns a generator,** a kind of iterators that you can iterate only once over! **Generators** do not store values in memory. eg. like range(1, 101, 1), used like return.

VARIABLE SCOPE

+ Local variable defined inside a function. can be accessed only inside that function.

+ Global variable can be accessed from anywhere in the program. Created outside the funct.

+ IMPORTANT **avoid using global variables in a function:**

(i) Its inefficient; Python first searches for a local variable with the function. if it doesn't find any, it then continues to search for a global variable with this name

(ii) Unwanted modifications in mutable objects, eg lists, may happen by mistake, and it will affect all the results

POSITIONAL & KEYWORD ARG'S

+ positional arg. Stored in LIST; Specified by position in argument list of the function eg: def funct(x, y): …

+ keyword arg. Stored in DICTIONARY, Specified by Keyword, but a keyword can be excluded, and the argument passed as positional arg.If keyword is included, no order required.

+ declaring arg's positional and keywords arg's can be mixed, positional arg's must be called at first.

*** Call by value ;** With Python, you operate on the copy of a variable when passing in the function.

+ Call by reference; When the function is working with mutable object eg: list, the values are called by reference to a given object, thus a mutable object may be chnaged See passin the variable

VARIADIC ARG'S

The function may take, none, one or many variadic arguments

***args** **Variadic; arg's;** given at the end of list of arg's in funct., stored in a list. If passed to that funct in a list it must be unpacked with "*" (see below)

```
var = [1,1,1] # this list can have any length!
def myFunc ( *args ):
    results = 1
    for i in args:
        results += args[ I ]
    return results
myFunc( *var ) # var unpacked with *
```

***kwargs** **Keyword; arg's;**

```
def ourOrders(**kwargs):
    for i, j in kwargs.items():
        print(f'{i}: {j}',end='; ')
```

+ 1st way or passing kwargs; as normal keword arg'sbut we can use unlimited number of them

```
ourOrders( PAWEL="milk",  ANNA="tea")
```

+ 2nd way or passing kwargs; by unpacked dictionary

To unpack dct, you use ** in front of it.

```
dct = {"PAWEL":"milk", "ANNA":"tea", "TOMMY":"coffe"}
ourOrders( **dct )
```

+ agrs before **kwargs.

```
def myFunct( *args, **kwargs ):
```

USEFULL

*** def func():**

```
print("hello world");
```

Often included in each progr. to ensure that everything Is well installed and your python is running correctly

*** def main():**

```
if __name__ == "__main__"
main()
```

To ensure that each line of code is read before execution because Py don't support forward declaration:

```
def main( ):
    kitten( )
    def kitten( ): print ( "miau" )
if __name__ == "__main__":
    main( )        # at the bottom of your script!
```

__name__; returns the name of a current module

"__main__"; string literal, reserved for main file. If my function. would be running as module this should be '__moduleName__' not '__main__'

*** Imposing keyword arg's ;** THIS WILL MAY SAVE YOUR COMPANY, YOUR WORK AND DEFINIATELY WILL SAVE YOU A LOT OF TIME

+ add "*" at the beginning of argument list, thus removing all positional args. This will impose using keys with each argument.

```
def myFunct(*, x=0, y=0, z=0):
    return x + y + z
myFunct(x = 1, y = 1, z = 1) # will no run with myFunct( 1, 1, 1)
```

PARAMETER PASSING

- **Name binding:** object exist independently from the name.
- **assigment between names** does not create a new objects
- **when re-bininding** the name to new variable, all other names bound to the original object are not affected
- **when calling myFunc(x) (see example 2), we create a new binding within a scope of that fucntion,** however the object remains the same, ie using name myFunc, we may affect a general variable.
- **only when calling myFunc(x) and changing immutable object, eg, tuple, it will create a new object** with a new name that is within the scope of myFunc() (see example 3)
- **if x is bound to mutable object, then any changes made in myfunc(x) will be reflected outside on x**

EXAMPLE 1 – NAME BINDING

```
x = 5
y = 10
z = x    #  new name for object  5
x = y    #  name "x" has been now re-binded to object with t
         #  z was not affected and it is still object 5
print("x =", x, ", y =", y, ", z =", z)
#         x = 10 , y = 10 , z = 5
```

NAME BINDING IN A FUNCTION

IMPORTANT

changes introduced in a function may affect the mutable objects such as list.

a) some function

```
myList = [1, 2, 3]
def myFunc(a):
        a.append(4) # “a” is only a name binding to “myList”
                    Thus this change will also affect
        a = [5, 6]      # Here we bind name “a” to new object
        return a        all previous changes introduced into
                        myList will  stay, but a will now refer
                        to new object.
```

b) run

```
newList = myFunc(myList)
```

c) see the results

```
print("myList: ", myList)
print("newList: ", newList)
#     myList:  [1, 2, 3, 4]
#     newList:  [5, 6]
```

he name "y", ie 10;

“myList” , as both of them bind to the

7. CLASS

NEW CLASS

```
class Car( ):
        def __init__(self, brand, car_type):
                self.brand = Brand
                self.car_type = Type
```

* **__init__;** constructor function, a special Python method called when you create new instance of a class, must be included

* **self;** the first arg. In __init__, must be included in a constructor def.and in each other function created within a class!

* **atributes;** Variables of __init__ funct. Must be given values, whenever you create new class instance. Can be empty. myCar = Car(“BMW”, “TR2.0” ; HenryCar = Car(“VW”, “Golf5”)$

* **no return in __init__** ; the constructor can not have return method! all returned values should be moved to other methods within that class!

PUBLIC & PRIVATE VARIABLES

https://radek.io/2011/07/21/private-protected-and-public-in-python

* **Variable encapsulation**; common practice in Java, C++, and Python. to make variables of a class private

+ **Protected Variable;** single underscore, eg: _varName, accessible only from within the class and it’s subclasses, just a naming convention.

* **Private variable;** double underscore, eg: __varName; should not be accessed from outside the class, causes name mangling

* **name mangling**; every member name prefixed with at least two underscores and suffixed with at most one underscore is changed into _<className><memberName>, ie. into private variable

+ example:

+ private variable is not given as parameter to __init__ funct.

```
# set new class
class Car(object):
    def __init__(self, brand, color):
        self.brand = brand      # public variable
        self._color = color     # protected variable,
        self.__age = None       # private variable, given by convention

    def give_age(self, age):
        self.__age = age

    def get_car_age(self):
        print(self.__age)

# create instance
myCar = Car("BMW", "blue")
myCar.brand             # ok
myCar._color            # ok
myCar.__age             #AttributeError

# change public or ptotected variables
myCar.brand = "VW"

# set & call private variable
myCar.give_age("13y")
myCar.get_car_age()     #13y
```

SINGLE CLASS INHERITANCE

https://thepythonguru.com/python-inheritance-and-polymorphism/

* **base / super / parent class**; a class that was created at first

* **subclass / derived / child class,** a class that inherits attributes and methods form its base class(es)

* **function overriding:** you may replace method from base class, defying new method in a child class with the same name and parameters.

```
# base class
class car():
    def __init__(self, brand, car_type):
        self.__brand = brand
        self.__car_type = car_type
    def get_brand(self):
        return self.__brand
    def get_car_type(self):
        return self.__car_type
# test
my_last_car = car("VW", "Golf"); my_last_car.get_brand()

# inherit car class in my_old_cars class
class my_old_cars(car):
    def __init__(self, brand, car_type, age):
        super().__init__(brand, car_type)
        self.__age = age
        # super(). calls __init__ method from superclass,
        # works only wiht one inherited class
    def return_brand(self):
        return super().get_brand()
        # super(). call all functions in base class, and provides their results
        # to a current method
    def return_car_type(self):
        return self.get_car_type()
        # self also works here, because all base class methods
        # are now inherited by the new class
my_first_car = my_old_cars("opel", "astra", "12y")
my_first_car.return_brand() # using method from child class
my_first_car.get_brand() # using method from base class
```

MULTIPLE CLASS INHERITANCE

```
class MySuperClass1():
    def method_super1(self):
        print("method_super1 method called")
class MySuperClass2():
    def method_super2(self):
        print("method_super2 method called")
class ChildClass(MySuperClass1, MySuperClass2):
    def child_method(self):
        print("child method")
c = ChildClass()
c.method_super1()
c.method_super2()
# you can run all inherited methods, and unless they do not override
each other, they can be called, at each time with new class.
```

11. DIRECTORIES AND FILES

OS MODULE

os.chdir(PATH **)** Set directory

os.getcwd( ) Current working directory

os.chroot(PATH **)** Change the root dir of the current process.

os.listdir(PATH **)** Return a list of the entries in PATH directory

os.mkdir(PATH **)** Create new directory

os.makedirs(PATH **)** Recursive directory creation function.

os.remove(PATH **)** Remove (delete) the file path. (FILE)

os.rmdir(PATH **)** Remove (delete) the directory path (DIR)

os.removedirs(PATH **)** Remove directories recursively.

os.rename("old_file_name", "new_file_name" **)** Rename

MAGIC COMMANDS, IPYTHON

+ Start with % or %%

%dir current dir stack, list all stuff we have in a memory:

%ls list of elements in current directory

%cd change current directory

%cp it is as in a system copy

%mkdir - :)

%mv - :)

%rm - :)

%rmdir - :)

FIND WHAT FILES YOU HAVE IN YOUR CURRENT DIRECTORY

glob.glob() simple; use standard unix notation:

***** - everything

? - one sign

[0-9]. - any one digit

[!a] - not "a"

https://facelessuser.github.io/wcmatch/glob/

Step 1. go to directory you need to search.,

Step 2. Search that dir with glob.glob()

```
import glob, os                    .
os.chdir("/Users/pawel/Desktop")
for file in glob.glob("*"):
        print(file)
```

os.walk() **May Provide info on all files in your computer!**

os.walk() generates the file names in a given directory by walking the directory tree either **top-down** or **bottom-up**

```
import os
os.chdir("/Users/pawel/Desktop")
for root, dirs, files in os.walk(".", topdown = False):
        for name in files:
                print( os.path.join( root, name ) )
        for name in dirs:
                print( os.path.join( root, name ) )
```

Comments:

```
# - root here is. "." this is from where we are starting
# - topdown =True//False changes order only
# - we print separately, file names and dir names
# - there is only one root ., - current dir, thus, you ,skip that in printing.
```

FIND FILES WITH MATCHING PATTERN

glob.glob( ) eg:

```
import glob, os
os.chdir("/Users/pawel/Desktop")
for file in glob.glob( "*.txt" ):
        print(file)
```

os.listdir( ) eg: all files with .rtf extension

```
import os
for file in os.listdir("/Users/pawel/Desktop"):
        if file.endswith( "rtf" ):
                print( os.path.join("/Users/pawel/Desktop", file))
```

os.walk() # e.g. find all txt files:

```
import os
PATH = "/Users/pawel/Desktop"

for root, dirs, files in os.walk(PATH, topdown = True):
        for file in files:
                file.endswith(".txt"):
                        print(os.path.join(root, file))
```

12. PRINT

*format

* .format
  - method of the string class
  - used to get variable in the string
  - sets order of variables imprinted string

```
x = 42
print("x is{}".format(x))

# or
x = 42
s = "x is{}".format(x)
print(s)
```

+ Multiple elements - Default Order

```
x,y = 42, 110
s = "x is{0}, and y is: {1}".format(x, y)
print(s)                                    #  x is42, and y is: 110

x, y = 8, 9
print("x and y are: {} {}".format(x,y))
#   x is42, and y is: 110
```

+ Multiple elements: Custom Order (* .format:)

```
x, y = 8, 9
print("{1}{0}".format(x,y), end="\n", flush = True)
#   98
```

+ Adjust spaces

```
x = 8
print(  "{ :>9}'".format(x)   )   #       9 spaces before x
print(  "{ :<9}'".format(x)   )   #       9 spaces after x
print(  "{ :>09}'".format(x) )   #       9 Zeros before x
print(  "'{ :<09}'".format(x) )   #       9 Zeros after x
```

f-string

* f"string {x}"

  THE PYTHONIC WAY

- python 3.6 and later !
- like .format, just as the other way :)

```
x = 42
print(f "x is: {x}")

a, b = 8,9
x = f'seven {a:<09} {b:<09}'
print(x)
```

+ When .format has an advantage over f-string:
  - when using special characters, and escape characters:

```
x = [1, 80, 109088, 12999955555922]
for i in x:
print("{:>9}".format(i))
#               1
#              80
#          109088
#       12999955555922
```

IMPORTANT WHEN USING PRINT

* end =
- as default is "\n"

```
print(x, end=";")
```

* flush = True
- Output to the screen
- flushed result must have an obj. with .write method, eg. str

```
print(x, end = ' ', flush = False)
#   sys.stdout will be used if nothing is found
print(*objects, s
        ep=' ', end='\n', file=sys.stdout, flush=False)
```

+ python 2 legacy code
- in py2 strings and print were not objects
- e.g.:

```
x = 42
print("this is from py2 %d" %x)
```

CAUTION py3 # mess with time date is still used !!!!!

13. PYTHONIC CODE

+ PEP 8

Guid-line how pythonic code should be written and formatted:

https://www.python.org/dev/peps/pep-0008/

+ add "*" at the beginning of argument list,

thus removing all positional args. This will impose using keys with each arguments.

```
def sum_Function(*, x=0, y=0, z=0):
        return x + y + z
sum_Function(x = 1, y = 1, z = 1)
#       3
```

+ USE UNPACKING WITH **

to constructing a dict, from other dictionaries:

```
dict_1 = {"id_1":1, "id_2":2}
dict_2 = {"id_1":5, "id_3":3}
dict_all = {**dict_1, **dict_2}
# dict_all:       {'id_1': 5, 'id_2': 2, 'id_3': 3}
```

Caution: the same key in both dict's, thus the second one will be taken

---

USE FUNCTION TO ENCAPSULATE IF ELSE LOOPS

```
x = 200
def test_x(*,x):
   if (x==0): return f"x is zero look:{x}";
   if (x<0): return f"x is less then zero look:{x}";
   if (x>0): return f"x is more than zero look:{x}";
..
test_x(x=200)
#       'x is more than zero look:200'
```

USE F'String (Python 3x)

```
x = 200
y = 1200
f"the score is equal to: {x+y+1}"
#       "the score is equal to 1401"
#       place any function of expression in { }
..
Caution: f'strings do not support "\n"!
you must use .format or simply print to get it:
names = ["Adam", "Marta", "Pawel"]
print(f"Names are:", *names, sep= "\n")
#       Names are:
#       Adam
#       Marta
#       Pawel
# Comment: * before "names" unpack that list
```

USE DICT TO PACK YOUR ITEMS

(not list)

+ ITEMS

```
import numpy as np
myArray = np.arange(1,10).reshape(3,3)
myList = list(range(1,10))
MyVariable = "myVariable is 1"
```

+ LIST: this way, dont tell what you have where

```
item_list = [MyVariable, myList, myArray]
```

+ DICT; we can search key much fasterm, without accessing the objects

```
item_dict = {"MyVariable": MyVariable,
            "myList": myList.copy(),
            "myArray": myArray.copy()}
item_dict.keys()
#       dict_keys(['MyVariable', 'myList', 'myArray'])
#       use .copy()
.       to ensure that dict. hold different objects
```